



Python - Les fonctions

[Mise à jour le : 30/6/2021]

- **Sources**
 - Python.org : [documentation](#).
 - [Fonctions natives](#) (built-in)
- **Ressources**
 - **Real Python** : [Defining Main Functions in Python](#)
- **Mots-clés** : fonction, paramètre, déclaration, appel, signature, docstring, indication de type, lambda, espace de nom.



Les mots ci-dessous sont dits “réservés”. Ils ne peuvent pas être utilisés comme nom de variable. Les mots soulignés sont une nouveauté de Python 3. Les mots en **gras** sont utilisés dans cette page.

and	continue	finally	is	raise
as	def	for	lambda	return
assert	del	from	<u>None</u>	<u>True</u>
<u>async</u>	elif	global	<u>nonlocal</u>	try
<u>await</u>	else	if	not	while
break	except	import	or	with
class	<u>False</u>	in	pass	yield

- [Fonctions natives](#) (**built-in**)¹⁾ utilisées dans les exemples : **print()**.

1. Généralités

- **Source** : définir des fonctions sur docs.python.org

En programmation, une fonction est un « **sous-programme** » permettant d'effectuer des opérations répétitives. Au lieu d'écrire le code complet autant de fois que nécessaire, on crée une fonction que l'on appellera pour l'exécuter, ce qui peut aussi alléger le code et le rendre plus lisible. [Wikiversité](#)



Pour utiliser une fonction, deux étapes sont nécessaires :

1. **déclarer** la fonction
2. **appeler** la fonction

1.1 Forme simple de la déclaration et de l'appel (ordonnée)

Dans cette forme, les arguments sont dit: "positionnels" car leur position lors de l'appel doit correspondre à la position des paramètres dans l'en-tête de la fonction.

*.py

```
# Déclaration
def nomfonction(parametre1, parametre2, ..., parametreN)
    """ Documentation """
    # bloc de code
    return valeur

# Appel
nomfonction(argument1, argument2, argumentN) # parametre1 <- argument1
etc.
```

Exemple

exfonc1.py

```
def cube(x):      # Déclaration de la fonction cube destinée à traiter le
                  # paramètre x
    return x*x*x

resultat = cube(2) # resultat = 8 après l'appel de la fonction cube
                  # avec comme argument la valeur 2
```

1.2 L'instruction return

L'instruction **return** renvoie la valeur calculée par une fonction. On peut ainsi l'affecter à une variable. Cette instruction arrête le déroulement de la fonction. Le code situé après **return** ne s'exécutera pas.



En Python, une fonction **peut renvoyer plusieurs valeurs**, séparées par une virgule, que l'on affecte à plusieurs variables également séparées par une virgule.

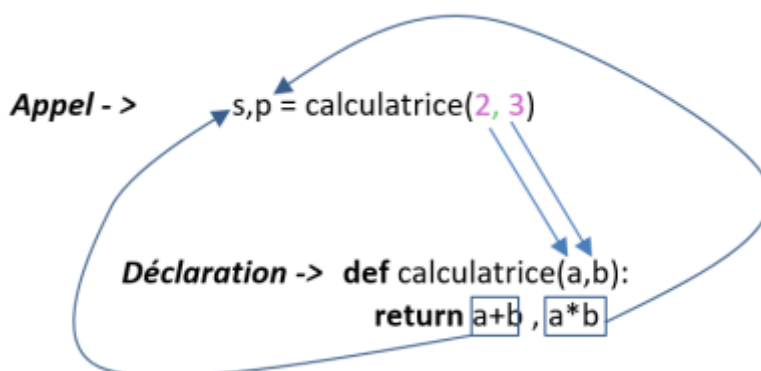
Exemple

*.py

```
def calculatrice(a,b):
    return a+b,a*b

s,p=calculatrice(2,3)
```

```
print('somme=',s,'produit=',p) # Résultat : somme= 5 produit= 6
```



1.3 L'instruction pass

Python ne disposant pas d'accolade pour délimiter les blocs de code, il existe une instruction **pass**, qui ne fait rien.



pass permet de définir une **fonction vide**.

Exemple

*.py

```
def foo():  
    pass
```

1.4 Signature d'une fonction



En Python, la **signature** d'une fonction est **son nom**.

Exemple

```
# Signature de table(nb, max=10)  
def table()
```



On ne peut pas surcharger les fonctions en Python.
Deux fonctions ne peuvent pas avoir le même nom.

1.5 Indication de type



Depuis **Python 3.5**, il est possible de fournir des indicateurs de type lors de la définition des fonctions.

Exemple

*.py

```
def mult2(x:int)-> int:
    return x*2

print(mult2(2)) # résultat : 4
```

1.6 La fonction main

'__main__' est le nom du scope dans lequel le code s'exécute en premier. **Le nom d'un module** (son `__name__`) vaut '**__main__**' lorsqu'il est lu de l'**entrée standard**, lorsque c'est un **script**, ou une **invite interactive**.

Un module peut découvrir s'il est exécuté dans le scope principal en vérifiant son `__name__`, ce qui permet typiquement d'**exécuter du code** lorsque le module est exécuté avec **python -m** mais pas lorsqu'il est importé :

Exemple

*.py

```
# Si le code ci-dessous est enregistré dans un fichier hello.py
# Il pourra être exécuté par python -m hello
def main():
    print("Hello World")

if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Résultat dans la console

```
phili@PC-BUREAU MINGW64 ~/OneDrive/Bureau/test
$ python -m hello
Hello World
```

Pour aller plus loin

- **Real Python** : [Defining Main Functions in Python](#)

2 Passage des arguments et appel de fonctions

2.1 Paramètres par défaut (optionnels)



On peut préciser une **valeur par défaut** pour les paramètres de la fonction.

exfonc2.py

```
# Définition
def table(nb, max=10): # si la valeur de max n'est pas précisée alors
    # max prend la valeur 10 par défaut
    """Fonction affichant la table de multiplication
    par nb de 1*nb à max*nb max*nb >= 0"""

    i = 0
    while i < max:
        print(i+1, "*", nb, "=", (i+1)*nb)
        i = i+1

# Appels
table(4,3) # max = 3
table(2)   # max = 10
```



La chaîne placée entre ' ' et ' ' est une **docstring**, elle est affichable par **help(nomfonction)**

Exemple

```
[8] help(table)
```

```
Help on function table in module __main__:
```

```
table(nb, max=10)
    Fonction affichant la table de multiplication
    par nb de 1*nb à max*nb max*nb >= 0
```



Les arguments peuvent être passés dans un **ordre quelconque** si on précise le nom des paramètres.

Exemple

exfonc3.py

```
# Définition
def fonc(a,b,c,d=12)

# Appel
print(fonc(b=6,a=1,c=4)) # Passage des arguments dans un ordre
quelconque
# Résultat : (1, 6, 4, 12), d conserve sa
valeur par défaut
```



Ne **pas** utiliser **d'objet mutable** pour les valeurs par défaut. A [voir](#).

Exemple

*.py

```
# ne faites SURTOUT PAS ça
def ne_faites_pas_ca(options={}):
    "faire quelque chose"

# mais plutôt comme ceci
def mais_plutot_ceci(options=None):
    if options is None:
        options = {}
    "faire quelque chose"
```

2.2 Liste et paramètres de fonction



Lorsque le **nombre de paramètres** d'une fonction est **inconnu**, on lui passe une liste en plaçant une étoile * devant le nom du paramètre destiné à recevoir la liste.

Exemple

*.py

```
def affiche(*parametres):
    print(f"J'ai reçu le texte : {parametres} en paramètre.")

affiche(12,14,16,18) # Résultat : J'ai reçu le texte : (12, 14, 16, 18)
en paramètre.
```

2.3 Dictionnaires et paramètres de fonction



On peut récupérer les **paramètres nommés** dans une fonction en plaçant **deux étoiles **** devant le nom du paramètre destiné à recevoir le dictionnaire.

Exemple

*.py

```
def affiche(**parametres_nommés):  
    print(f"J'ai reçu le texte : {parametres_nommés} en paramètres  
    nommés.")  
  
affiche(a=12,b=14,c=16,d=18) # Résultat : J'ai reçu ce texte : {'a':  
12, 'b': 14, 'c': 16, 'd': 18}  
                             # en paramètres nommés
```

2.4 Synthèse

Lors de la **définition de l'en-tête** d'une fonction, les 4 familles de **paramètres** que l'on peut déclarer sont :

1. paramètres positionnels (usuels),
2. paramètres nommés (forme *name=default*)
3. paramètres **args* qui attrape dans un tuple le reliquat des arguments positionnels
4. paramètres ***kwargs* qui attrape dans un dictionnaire le reliquat des arguments nommés



Lors de l'**appel** de la fonction, les **arguments** peuvent être :

1. ordonnés,
2. nommés,
3. passés avec la forme *** (tuple unpacking)
4. passés avec la forme **** (dictionnaire)

3. Portée des variables

Les **arguments** passés dans les fonctions sont des **variables locales**. Elles n'existent qu'à l'intérieur de la fonction contrairement aux **variables globales** déclarées dans le programme principal. Pour plus d'informations, voir "[Variables, types numériques et entrées / sorties dans la console](#)"

4. Les fonctions lambda

Python permet de définir des **mini-fonctions** sur une ligne. Empruntées à Lisp, ces fonctions dites **lambda** peuvent être employées partout où une fonction est nécessaire.



Une fonction lambda est une fonction qui prend un nombre quelconque d'arguments (y compris des arguments optionnels) et **retourne la valeur d'une unique expression**.

- Syntaxe

*.py

```
lambda arg1,arg2,...:instruction de retour
```

- Exemple1 : lambda affectée à une variable

```
f=lambda x:x*x  
f(3) # renvoie 9
```

- Exemple 2 : lambda passée en paramètre à une fonction

*.py

```
def image(f):  
    for x in range(-1,10):  
        print(f"{f(x)} : {x}")  
  
image(lambda x:x**2 -1) # Résultat : 0 : -1  
                        #           -1 : 0  
                        #           0 : 1 etc.
```



Les fonctions lambda ne peuvent ni contenir des commandes ni contenir plus d'une expression.

5. Les fonctions natives (built-in)

- Source : [fonctions natives](#) sur python.org



L'interpréteur Python propose quelques **fonctions** et types natifs qui sont **toujours disponibles**.



La liste et la description sont accessibles à partir du lien ci-dessus.

Résumé

- Une fonction est une portion de code contenant des instructions, que l'on va pouvoir réutiliser facilement.
- Découper son programme en fonction permet une meilleure organisation.
- Une fonction commence par le mot-clé **def**.
- Les fonctions peuvent recevoir des **paramètres** et renvoyer une ou plusieurs informations grâce au mot-clé **return**.
- Une fonction n'a pas forcément de paramètres;
- Une fonction peut appeler une autre fonction à condition que la fonction appelée contienne un return et est été définie avant la fonction appelante.



Pour aller plus loin

[How to Use Python Lambda Functions](#)



Quiz

- [Python Lambda Functions Quiz](#)

1)

Fonctions toujours disponibles.

From:

<https://webge.fr/dokuwiki/> - **WEBGE Wikis**

Permanent link:

<https://webge.fr/dokuwiki/doku.php?id=python:bases:fonction&rev=1628697600>

Last update: **2021/08/11 18:00**

